一、作用域

继承与原型链 - JavaScript | MDN (mozilla.org)

变量分类

1. 全局变量：顶层代码里面声明的变量；
2. 局部变量：

- 函数内部声明的变量是局部变量；
- 函数的参数也是局部变量，是函数内部的全局变量；
- 只能在函数内部使用；

```
var a = 200;
    console.log(a);//200
    function  fn(){
    /**
    函数内部声明的变量是局部变量，
    函数外面不可直接调用*/
 var b = 300;
     console.log(a);//200
     console.log(b);//300
  //我不声明，直接赋值:坚决不推荐这么做
        c = 400;
  }
  fn();
     console.log(c);//400
     console.log(b);// b is not defined
```

3. 块级作用域：ES6（暂时不管）
4. 变量的解析有个就近原则；

变量提升

- 使用var声明的变量会提升到前面：只提升声明，不提升赋值；

```
console.log(a); //undefined
var a = 2;
```

解释：当 var a = 2;时，只是一个简单的声明语句。但 JavaScript 实际上会将其看成两个声明： var a; 和 a = 2;。第一个定义声明是在编译阶段进行的，第二个赋值声明会被留在原地等待执行阶段。

JavaScript**处理形式：**

```
var a;
console.log(a);
a = 2;
```

注：无论作用域中的声明出现在什么地方，都将在代码本身被执行前首先进行处理。可以将这个过程形象地想象成所有的声明（变量和函数）都会被“移动”到各自作用域的最前端，这个过程被称为 变量（函数）提升。

函数提升

- 使用function声明的函数会提升到前面：整个函数（包括函数体）都会提升到前面

```
var a = true;
foo();

function foo() {
 if(a) {
        var a = 10;
 }
    console.log(a); //undefined
}

fun1(); // fun1...
fun2(); // UncaughtTypeError: fun2 is not a function
// 函数声明，会被提前创建
function fun1(){
    console.log("fun1...");
}
// 函数表达式，不会被提前创建（变量会被提前声明，但函数不会被提前创建）
var fun2 = function(){
    console.log("fun2...");
}
```

JavaScript执行过程：

```
function fn() {
 var a;
 if(a) {
        a = 10;
 }
    console.log(a); //undefined
}
//var a = 10;会被JavaScript分为var a; var a会被提升到函数作用域中的最顶端,成为局部变量 a;
//在fn(...) {} 的函数作用域中，这个重名局部变量a 会屏蔽全局变量a，，
//换句话说，在遇到对a的赋值声明之前在 fn(...) {}中a 的值都是 undefined
//所以一个 undefined 的 a 进入不了 if(a) {...} 中，所以最后被打印出来的是 undefined。


var a;
a = true;
//var声明的变量会提升到前面，
//只提升声明， 不提升赋值；
//function声明的函数也会提升到前面
 fn();
```

解释： fn(...) {} 的位置被移到了 fn();的前面（整个函数（包括函数体）都会提升到前面），根据前面对变量提升的解释，可以很容易理解这是函数发生了提升，这就是函数提升

- 函数声明会被提升，但表达式函数**部分**不会提升，函数表达式不会被提升

```
var a = true;
fn();
var fn = function() { //TypeError: fn is not a function
 if(a) {
        var a = 10;
 }
    console.log(a);
}
```

JavaScript执行过程:

```
var a;
var fn;
```

```
3
4 a = true;
5 fn();
6
7 fn = function() { //=后的赋值部分不会提升，在执行时才会生成
8  if(a) {
9         var a = 10;
10  }
11     console.log(a);
12 }
```

- 函数内部也存在自己的变量提升

```
1 var a= 500;
2    function  fn(){
3 /* 函数内部也存在自己的变量提升*/
4     console.log(a);//undefined
5 var v= 600;
6 var d = 900;
7 var a = 200;
8    function f1(){
9     var a = 900;
10    console.log(a);//900
11    }
12     f1();
13 }
14   fn();
```

JS代码的执行步骤

三步：语法检查、预编译、执行语句

```
1 /*
2 1，就近原则：作用域链，先AO，后GO；
3 2，变量提升：只提升声明，不提升赋值；
4 3，JS代码执行：语法检查、预编译、执行语句
5 */
6 var x = 1;
7     function ScopeTest() {
```

```
    console.log(x); //undefined
    var x = 'hello world';
    console.log(x); // hello world
  }
   ScopeTest();
```

语法检查：括号是否配对、标识符是否合法、循环语句是否遵守语法；从上到下进行 词法分析、语法分析、语义分析 等处理

```
/**
1,语法检查：通过；
2，预编译：
   全局预编译：
    GO{
       getName:function getName() { console.log(5); }
       --->
       function () { console.log(4); }
        }
3，执行语句：
     =是赋值语句
*/

var getName = function () {
    console.log(4);
   }
function getName() {
    console.log(5);
   }
   getName();

/*
AO{
    this
  }
*/
```

预编译：执行上下文的创建，包括创建变量对象、建立作用域链、确定 this 指向等

- **全局预编译：**

1. 编译script标签内嵌或外联的顶层代码；

2. 过程：

- 创建一个GlobalObject，简称GO，执行期上下文（场景）；
- 变量提升：放到GO里面

  var声明的变量会提升到前面：只提升声明，不提升赋值；

  写代码的时候，尽可能规避变量提升；

3. 函数的提升：顶层代码里面声明的函数是全局的

- **函数预编译**：

1. 发生在函数调用的前一刻；
2. 过程：

- 创建一个对象ActiveObject：AO 活动的
- 所有的函数在每一次调用的时候，都会产生一个独立的AO；

```
/**
GO:{
    window,
    a:undefined-->200,
    fn:function(){var b = 300;console.log(b);}
  }
*/

    console.log(a); //undefined
   var a = 200;
       console.log(a); //200
     function fn(n) {
       console.log(n);
       var b = 300;
       console.log(b);
       console.log(a);
     }
    fn(200); //产生一个AO：ActiveObject
        //200
        //300
        //200
    fn(2000); //当再一次调用该函数，会产生一个新的AO：ActiveObject
        //2000
        //300
        //200
```

- 把函数内部声明的变量以及形参放到AO里面；

- 如果在函数内部找到声明的函数，也放到AO里面；
- 函数内部也会存在自己的变量提升；

```
/*
GO:{
   num:undefined  --->5,
   func1:f(){ 函数代码 }
   }
*/

var num = 5;
 function func1() {
     var num = 3;
     var age = 4;
     function func2() {
         console.log(num);//1,undefined
     var num = 'ivan';
     function func3() {
        age = 6;
      }
    func3();
//AO: {}

   console.log(num);//2,ivan
   console.log(age);//3,6
  }
 func2();
 /**
  AO: {
      num:undefined--->ivan,
      func3:f(){函数体}
     }
  */
}

func1();
/**
  AO:{
     num:undefined--->3,
```

```
    age:undefined----》4----》6,
    func2:f(){函数体}
}
*/
```

作用域链：

通过外部词法环境的引用，作用域可以顺着栈层层拓展，建立起从当前环境向外延伸的一条链式结构

1. [[scope]]属性

- 作用域链属性；
- 给JS解释器使用的；
- 我们在写JS代码的时候不会直接去访问他；

2. 基于作用域链的变量查询：

- 当某个变量无法在自身词法环境记录中找到时，可以根据外部词法环境引用向外层进行寻找，直到最外层的词法环境中外部词法环境引用为null

执行语句：赋值语句、函数调用语句、循环语句等的执行

1. 这些都可以在调试工具里面观察到：

等号=:

```
/**
    前面(): 里面是个函数声明的表达式
    后面(): 自运行函数;
*/

(function () {
/*
1，在函数内部声明了一个变量a，是局部变量，只能在函数内部访问;
2，如果想一次性声明多个变量，需要用逗号（,）分开;
3，这里给变量b赋值为3，但是并没有声明变量b，所以会在window对象身上追加一个属性b;
*/

var a = b = 3;  //var a = 3, b = 3;  二者的区别，是不一样的
})();

//=: 变量赋值
//==:相等判断，不考虑类型    123  == '123'    true
//===:完全相等，考虑类型    123 === '123'    false
//!=: 不相等，值不相等  123  != '123'        false
//!==:  不完全相等，要么类型不同，要么值不同，如果都相同就是false
```

```
    console.log("a defined? " + (typeof a !== 'undefined')); //false
/* 解释：typeof a 是'undefined'，因为a在全局里面没有定义 */
// a defined? false
    console.log("b defined? " + (typeof b !== 'undefined')); //true
/** 解释：typeof b 是 'number'，因为b在函数内部没有声明，是一个全局的变量*/
//b defined? true
    console.log(b);//3
    console.log(typeof a);//undefined
```

二、类和对象

- 类：是对事物的抽象；体现在JS上，就是构造函数来描述一类事物的属性（特征）和方法（动作）
- 对象：是类的实例化；真实存在的；
- JS里面一切皆为对象；使用对象描述万物；

构造函数

1. 构造函数就是函数，是需要使用function来声明的；
2. 构造函数首字母大写；
3. 是用new来调用构造函数创建一个实例化对象；
4. 在构造函数里面，使用this指向构造函数创建的实例化对象；
5. 构造函数的默认返回值就是this；

```
function Person(name, gender, age){
    this.name = name;
    this.gender = gender;// this['gender'] = gender;
    this.age= age;
    this.say = function(){ console.log(`我是${this.name}`); }
   }
 var p1 = new Person('陈1', 'male', 19);
     console.log(p1);

 var p2 = new Person('陈2', 'female', 19);
     console.log(p2);

     console.log(p1.say);
     console.log(p2.say);
     console.log(p1.say == p2.say);
```

执行结果：

原型和原型链

原型：

原型:js里面一切皆可以看做对象，每个对象都有自己的原型，实例化对象的原型__proto_指向的是构造函数的原型prototype，构造函数的原型对象prototype 的原型_proto_ 指向的是 Object的原型prototype，object的原型prototype的原型__proto_指向null

原型链：

当读取实例的属性时，如果找不到，就会查找与对象关联的原型中的属性，如果还找不到，就去找原型的原型，一直到最顶层（__proto__为null）为止

当访问一个对象的属性（包括方法），一直到最后没找到就会报错。

1. 所有的对象都有一个属性__proto__（前后都是双下划线），我们称之为原型；
2. 构造函数有一个特有的属性prototype，用来定义构造函数的原型；
3. 我们在访问一个对象的属性或者方法时，如果他自身没有这个属性或者方法，就去自己的原型(__proto__)上面去找，如果没有找到，会继续顺着__proto__的__proto__继续去找，直到找到为止；（当然，我们在访问原型之上的属性或者方法的时候，不用加__proto__）
4. 实例化对象的原型(__proto__)指向的是构造函数的prototype；

```
var arr1 = [1, 2, 3];
//__proto__ 和 prototype的关系： 实例化对象的__proto__指向的是构造函数的prototype
console.log(arr1.__proto__ === Array.prototype); //true

//构造函数本身就是一个实例化对象：Array就是一个对象，是内置构造Function的实例化对象
console.log(Array.__proto__ === Function.prototype); //true

```

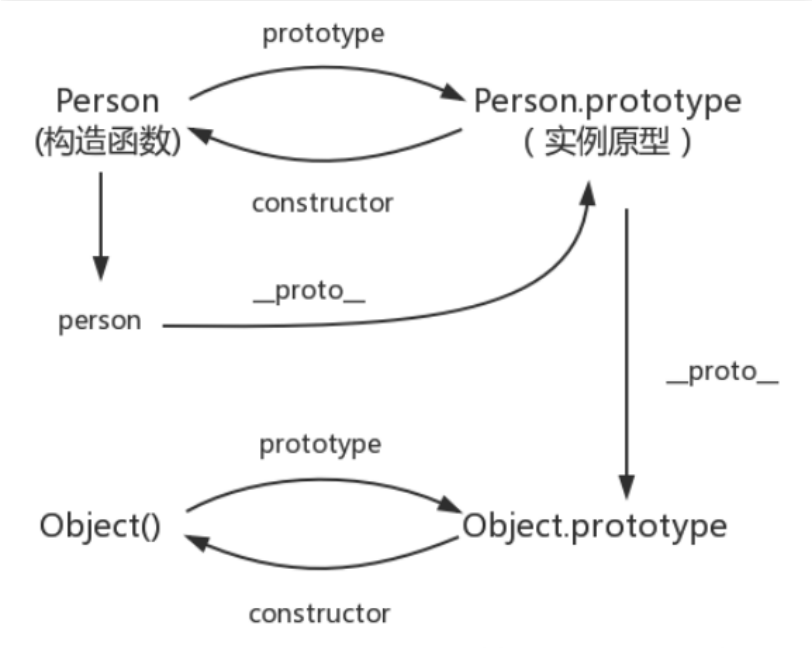
```
//构造函数Object是一个对象，所以有__proto__属性，指向Function的prototype
console.log(Object.__proto__ === Function.prototype); //true

//内置构造Function也是一个对象，所以有__proto__属性，指向Function的prototype
console.log(Function.__proto__ === Function.prototype); //true

//Array.prototype 也是一个对象，所以有__proto__属性，指向构造函数Object的prototype
console.log(Array.prototype.__proto__ === Object.prototype); //true

//Function.prototype 也是一个对象，所以有__proto__属性，指向构造函数Object的prototype
console.log(Function.prototype.__proto__ === Object.prototype); //true

//Object.prototype 也是一个对象，自然有__proto__
console.log(Object.prototype.__proto__ === null); //true
```



5. 所有的构造函数本身就是一个实例化对象，所以也有自己的__proto__，他指向的是内置的构造函数Function的prototype；

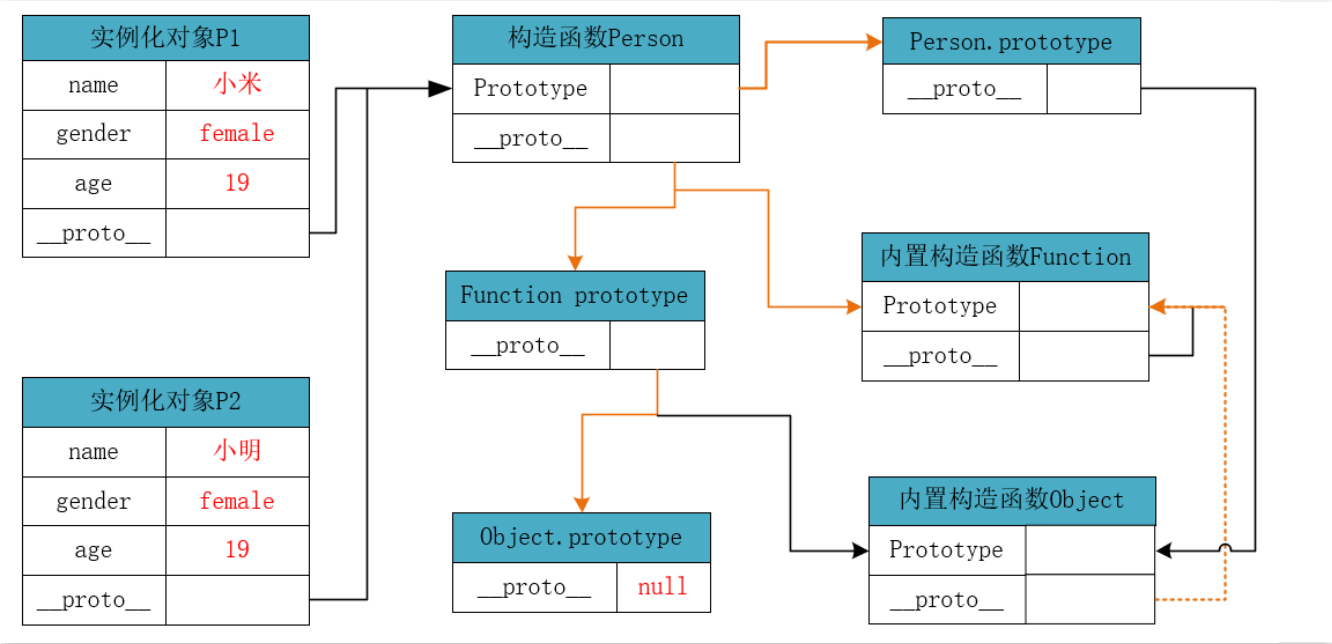
```
function Person(name, gender, age){
   this.name = name;
   this.gender = gender;
   this.age= age;
}
 //通过构造函数的特有属性prototype来定义构造函数的方法
Person.prototype.say = function(){
```

```
        console.log(`我是${this.name}`);
}
Person.prototype.run = function(){
        console.log(`我在跑步`);
}
var p1 = new Person('小米', 'female', 19);
    console.log(p1.__proto__ === Person.prototype); //true
var p2 = new Person('小明', 'female', 19);
    console.log(p2.__proto__ === Person.prototype); //true
    console.log(p1.say === p2.say); //true

    console.log(Person.prototype.__proto__  === Object.prototype); //true

    console.log(Person.__proto__ === Function.prototype);//true
    console.log(Object.__proto__  === Function.prototype);//true
    console.log(Function.__proto__ === Function.prototype);//true
    console.log(Function.prototype.__proto__ === Object.prototype);//true

    console.log(Object.prototype.__proto__  === null);//true
```



6. 对象中基于原型链的查找

- 原型：每一个JS对象（null 除外）在创建时就会与另一个对象关联，这个对象就是我们说的原型。每一个对象都会从原型中“继承”属性。
- **1当读取实例的属性时，如果找不到，就会查找与对象关联的原型中的属性，如果还找不到，就去找原型的原型，一直到最顶层（__proto__为null）为止**
- **2当访问一个对象的属性（包括方法）时，首先查找这个对象自身有没有该属性。如果没有就查找**

它的原型（也就是 proto 指向的 prototype 原型对象）。如果还没有就查找原型对象的原型（Object）的原型对象）。依此类推一直找到 Object 为止（null）。没找到就会报错

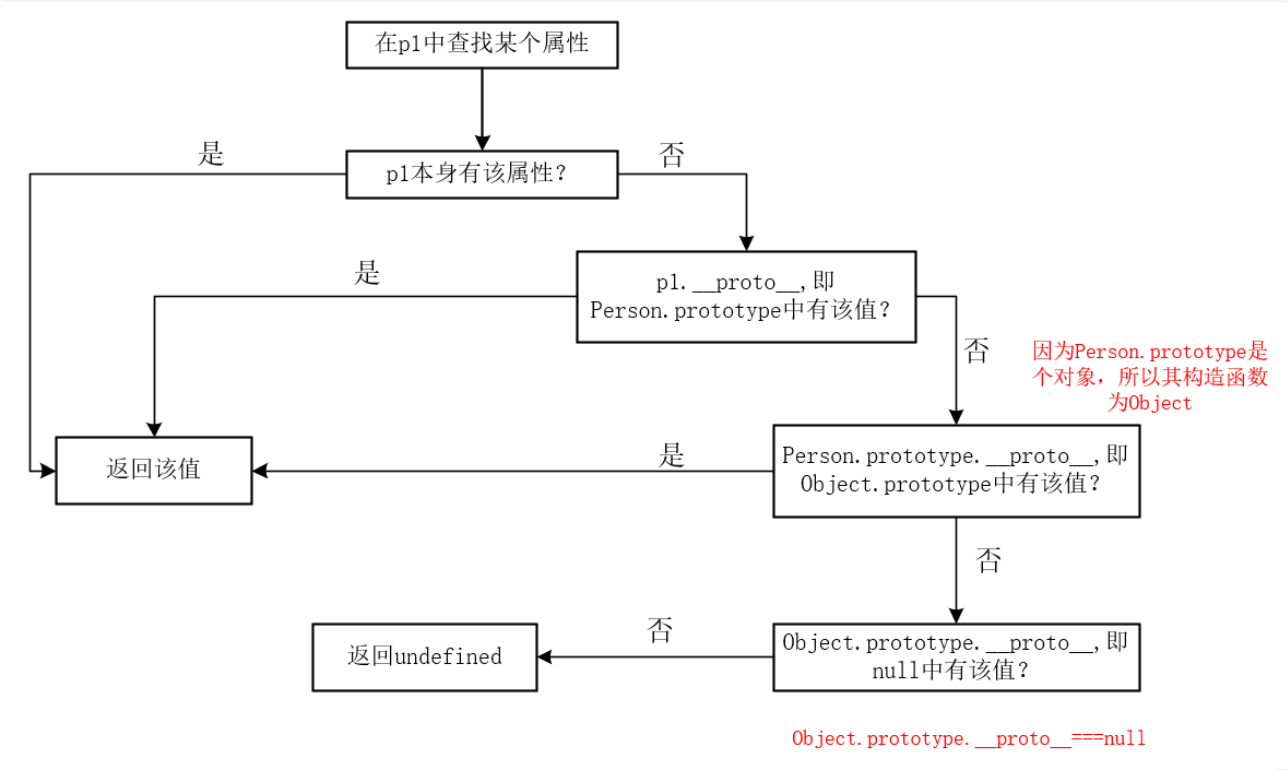
- **proto** 对象原型的 意义就在于为对象成员查找机制提供一个方向，或者说一条路线

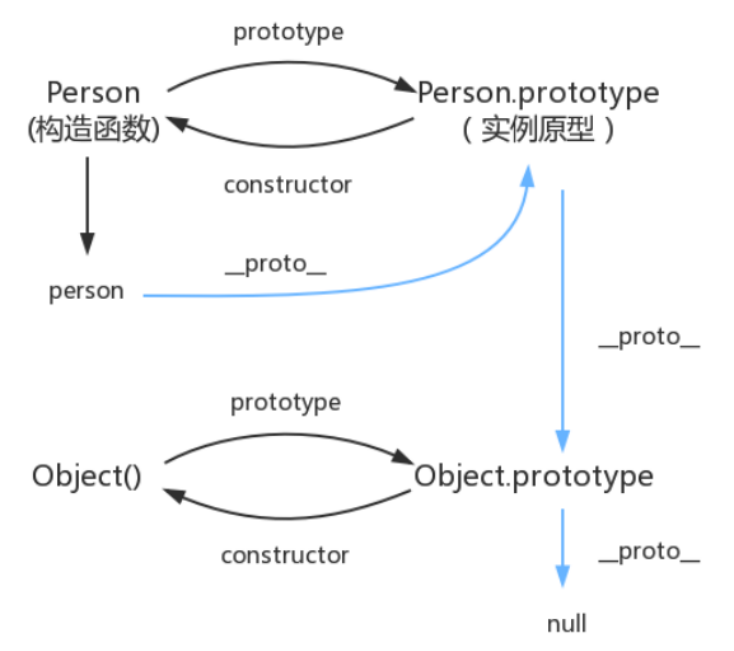
```
function Person(name, gender, age){
    this.name = name;
    this.gender = gender;
    this.age= age;
}

var p1 = new Person('小A');

console.log(p1.name); // 小A

console.log(p1.Name); // undefined
```





7. 原型给了我们一个进行属性或者方法扩展的机会；注意：数组和字符串内置对象不能给原型对象覆盖操作 Array.prototype = {} ，只能是 Array.prototype.xxx = function(){} 的方式。

```
通过原型对构造函数进行方法的扩展：
Array.prototype.max = function(){
 /**初始化为最小值*/
var max = -Infinity;
 /**这里用this拿到当前构造函数的实例化对象*/
 for(var i=0; i < this.length; i++){
    if(this[i] > max){
       max = this[i]
       }
    }
     return max;
}

var arr1 = [1, 200, -8, 0];
    console.log(arr1.max());//200

var arr2 = [100, 800, 900];
    console.log(arr2.max());//900

静态方法的定义：不要写到原型之上
var arr2 = [100, 800, 900];
```

```
Array.myFunMax = function(arr){
var max = -Infinity;
    for(var i=0; i < arr.length; i++){
       if(arr[i] > max){
          max = arr[i]
        }
     }
      return max;
}
    console.log(Array.myFunMax(arr2));//900
    console.log(Array.myFunMax([1,2,3,4,56,789]));//789
```

8. 可维护性可扩展性特别好
9. 原型链与作用域链的区别

- 原型链是通过 prototype 属性建立对象继承的链接；而作用域链是指内部函数能访问到外部函数的闭包

this指向和原型链

```
    function obj(name) {
            console.log(this);
            if (this === window) { //判断通过何种方式调用。
                if (name) {
                    this.name = name;
                } else {
                    this.name = 'name1';
                }
                return this;
            }
        }
        obj.prototype.name = "name2";

        var a = obj("name1"); //如果通过函数方式调用,this会指向window。
        var b = new obj(); //如果通过new方式调用,this会指向实例化后的对象,obj{}
        var c = obj(); //如果 函数调用的时候不带参数,默认name为name1
        console.log(a.name);//name1
        console.log(b.name);//name2
        console.log(c.name);//name1
```